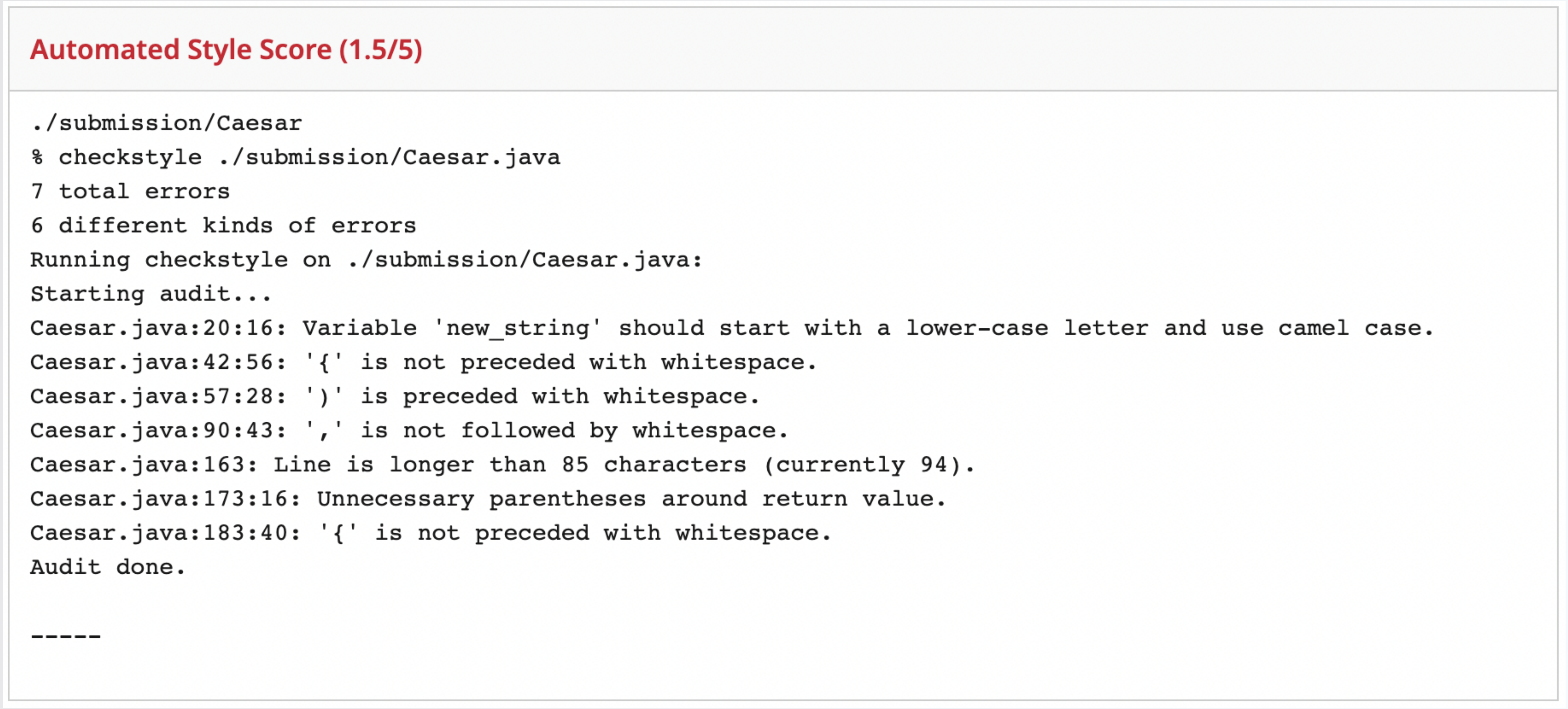
Records & Data Oriented Programming

Reminder about Autograder Output

- Gradescope has been reconfigured to automatically deduct points for style deductions
- Take a look sometime today to make sure that you have no automatic style errors!
- Other things that are **your responsibility to check for:**
  - submitting all files
  - compilation issues

If you see 0/40, that's not something you should ignore!



Automated Style Score (1.5/5)

```
./submission/Caesar
% checkstyle ./submission/Caesar.java
7 total errors
6 different kinds of errors
Running checkstyle on ./submission/Caesar.java:
Starting audit...
Caesar.java:20:16: Variable 'new_string' should start with a lower-case letter and use camel case.
Caesar.java:42:56: '{' is not preceded with whitespace.
Caesar.java:57:28: ')' is preceded with whitespace.
Caesar.java:90:43: ',' is not followed by whitespace.
Caesar.java:163: Line is longer than 85 characters (currently 94).
Caesar.java:173:16: Unnecessary parentheses around return value.
Caesar.java:183:40: '{' is not preceded with whitespace.
Audit done.

-----
```

Exam Reminders

- Plan to take your exam during the section for which you are registered.
- Take a practice exam once you're done with *Caesar*.
- All students who require SDS accommodation to take the exam should schedule their exam through the Weingarten Testing Center ASAP.
    - Any time on February 26th is acceptable.
- The exam only covers material up until functions & searching. Material covered today and Friday will be covered on HW04 and Exam 2.

Records & Data Oriented Programming

Background: Programming Paradigms

The different ways that we write & organize code are referred to as *programming paradigms*.

- For example, we've been using *imperative programming* techniques in Java so far
- Different varieties of problems we want to solve call for different ways of thinking about solutions!

Data Oriented Programming Writ Large

- A model of writing programs that separates the *code* from the *data*
- Works with immutable (unmodifiable) constructs in a program to focus on analysis & transformation of data
- Often an essential mindset for programs intended to make full use of hardware capabilities
    - scientific programming, graphics, video games

Disclaimer: This is just a way of thinking about what we're doing—it's not a binding set of rules!

Data Oriented Programming in CIS 1100

As novice programmers, we aren't going to worry yet about questions of efficiency and memory organization.

However, within the last few years, Java has included some constructs that allow us to separate *code* from *data* nicely.

Data Oriented Programming in CIS 1100: The Old Way

Imagine that we want to write a program that models planets in a planetary simulation.

- To do so, we need information about the position (x, y) and speed (x, y) of each planet, along with its mass and the name of some image with which to represent it
- Previously, we had to use cumbersome programming techniques like *parallel arrays* to do this😢

```
double[] px = new double[n];
double[] py = new double[n];
double[] vx = new double[n];
double[] vy = new double[n];
double[] m = new double[n];
String[] img = new double[n];
// for what it's worth, this technique is still useful sometimes
```

Record Types in Java: The New Way

- Similar to structs in C or records in OCaml
- Associate multiple pieces of data with the same entity to allow convenient access to all pieces at once
- *Immutable*
- The difference is sometimes elided when speaking about them, but:
    - *Record type* refers to the category of entities that are described using the same pieces of information
    - *Records* themselves are individual values in our program that model one individual instance of an entity belonging to this type

The Big Idea

Most real or imaginary world entities have multiple properties

In this module, we will learn how to represent the properties (or attributes) the entities that our program will manipulate

Example:

- Entity: student
- Properties: name, age, gpa, pennkey, graduation year
- Types: ….

The Big Idea

Most real or imaginary world entities have multiple properties

In this module, we will learn how to represent the properties (or attributes) the entities that our program will manipulate

Example:

- Entity: student
- Properties: name, age, gpa, pennkey, graduation year
- Types: String, int, double, String, int/String

Learning Objectives

To be able to define **record types**

To be able to initialize a **record** of a given type

Modeling with Records

Record types are used to model real-world entities.

A **record type** has some **properties** that can be used to describe all entities that belong to this type

- Properties can be used to describes a particular **record**
- All records of a type have the same properties but **not** the same values for those properties.

Records in Java

Record types are defined using a record definition, and they give a template for creating records

Record Type	Record
Cat	Garfield the cat
Cat	Izzy
Cat	Digby

Defining a Record Type

As simple as:

```java
public record RecordTypeName(type0 arg0, type1 arg1, ...) {}
```

That's it!

Defining a Record Type: Example

In my program, I want to model **points** in 2D space.

```java
public record Point(double xPos, double yPos) {}
```

Using the Record Type

We'll come back to the finer points of the new syntax here shortly, but for now:

```
public record Point(double xPos, double yPos) {} // defines the type

// expects an array of points as input
public static void drawAllPoints(Point[] points) {
    PennDraw.setPenRadius(0.01);
    for (int i = 0; i < points.length; i++) {
        // each value in the points array has type Point
        Point thisPoint = points[i];
        System.out.println("Drawing " + thisPoint); // Point can be printed
        // access attibutes with . notation
        PennDraw.point(thisPoint.xPos(), thisPoint.yPos());
    }
}
```

Defining a Record Type: Practice

In my program, I want to model **students** that each have a name, age, GPA, PennKey, & graduation year.

Defining a Record Type: Practice

In my program, I want to model **students** that each have a name, age, GPA, PennKey, & graduation year.

```java
public record Student(String name, int age, double GPA,
                      String pennKey, int graduationYear) {}
```

Defining a Record Type: Practice

In my program, I want to model **planets** that each have positions (x, y), velocities (x, y), mass, and images stored in files.

Defining a Record Type: Practice

In my program, I want to model **planets** that each have positions (x, y), velocities (x, y), mass, and images stored in files.

```
public record Planet(double xPos, double yPos,
                     double xVel, double yVel,
                     double mass, String imageFileName) {}
```

Creating a Record in Java

Each record type has a **constructor** that is used to **initialize** the **attributes** in a newly created record

The constructor and the record type have the same name

To create a record, you write

```java
RecordType variableName = new RecordType(arguments);
```

Constructors: More Detail

Creating a new record of a type requires us to call the **constructor** with the `new` keyword

- `new Cat("Garfield the cat", "orange")` calls the constructor for the `Cat` type.

For now, when you call the constructor, you need to pass in one value as input for each of the attributes that make up a record of this type.

- `new Cat()` or `new Cat("Garfield the cat")` or `new Cat(3.4, 7)` will all fail to compile!

Creating a Record in Java

Example:

A cat has the following attributes: name, color, so its record type may be defined with:

```
public record Cat(String name, String color) {}
```

To create a new orange Cat named “Garfield the cat” you write:

```
Cat garfield = new Cat("Garfield the cat", "orange");
```

Creating Records

We can create more than one record of the same type!

```
Cat garfield = new Cat("Garfield the cat", "orange");
Cat myCat = new Cat("mona", "yellow");
Cat yourCat = new Cat("midnight", "black");
```

Our program will manipulate the following records (cats):

Object name	name	Color
garfield	Garfield the cat	orange
myCat	mona	yellow
yourCat	midnight	black

Printing Records

Simple as calling `System.out.println()`!

```java
public record Point(double x, double y) {}
```

then, in main...

```java
Point a = new Point(0.5, 0.5);

System.out.println(a);
//prints Point[x=0.5, y=0.5]
```

Accessing Properties

Access the properties of a given record using the name given to the property as a function on that record:

```
public record Point(double x, double y) {}
```

then, in main...

```
Point a = new Point(0.5, 0.5);
double myX = a.x();
double myY = a.y();
```

Records Are Immutable Always

Trying `f.x = 5;` or `b.y = 7;` raises a compiler error independently of where the record is defined.

- Although `x` is the name of a property of `f`, its value cannot be changed once created.
- Java will explicitly prevent this and complain: `"cannot assign a value to final variable x"`

Comparing Records

Two Records can be compared to see if they contain exactly the same data by using `.equals()`

```
public record Point(double x, double y) {}
```

then, in main...

```
Point a = new Point(0.5, 0.5);
Point b = new Point(0.5, 0.5);
Point c = new Point(0, 0);

System.out.println(a.equals(b)); // true!
System.out.println(a.equals(c)); // false!
```

Where to Place the Record Definition

Defining the Record Within a Class

You can place the record definition *inside the class* but *outside of any other function.*

```java
public class Records {
    public record Foo(int x) {}

    public static void main(String[] args) {
        Foo f = new Foo(4);
        System.out.println(f.x()); // preferred way of accessing properties
        System.out.println(f.x); // this is allowed here, but not good
        f.x = 5; // compiler error!!!!
    }
}
```

In this example, accessing `f.x` is allowed. Assigning to `f.x` is not.

Defining the Record In Its Own File

```
// in Bar.java
public record Bar(int y) {}
```

```
// in Records.java
public class Records {
    public static void main(String[] args) {
        Bar b = new Bar(7);
         // this is the only allowed way of accessing property y
        System.out.println(b.y());
        System.out.println(b.y); // this is not allowed here
        b.y = 5; // compiler error, as before.
    }
}
```

In `Records.java`, accessing `b.y` is not allowed. Assigning to `b.y` is also not allowed.

Within an Existing Class vs. In Its Own File

You can put the record definition in either place.

- You should always access record properties using the **function syntax**
    - e.g. always prefer `f.x()` over `f.x`
- Why?
    - `f.x()` always works to access property `x` independent of your choice
    - It's best not to think of `f.x` as a variable anyway since its value can't actually change!